The Basics of UNIX/Linux

12-1. Dynamic Memory Allocation

Instructor: Joonho Kwon

jhkwon@pusan.ac.kr

Data Science Lab @ PNU

datalab

data science laboratory

This file is based on slides from Justin Hsia.

Lecture Outline

- Heap-allocated Memory
  - `malloc()` and `free()`
  - Memory leaks
  - Sample codes

Memory Allocation So Far (1/2)

- So far, we have seen two kinds of memory allocation:

```c
int counter = 0;     // global var

int main(int argc, char** argv) {
  counter++;
  printf("count = %d\n",counter);
  return 0;
}
```

- `counter` is ***statically***-allocated
  - Allocated when program is loaded
  - Deallocated when program exits

Memory Allocation So Far (2/2)

- So far, we have seen two kinds of memory allocation:

```
int foo(int a) {
  int x = a + 1;      // local var
  return x;
}

int main(int argc, char** argv) {
  int y = foo(10);    // local var
  printf("y = %d\n",y);
  return 0;
}
```

  - `a`, `x`, `y` are ***automatically***-allocated
    - Allocated when function is called
    - Deallocated when function returns

Dynamic Allocation

- Situations where static and automatic allocation aren’t sufficient:
  - We need memory that persists across multiple function calls but not the whole lifetime of the program
  - We need more memory than can fit on the Stack
  - We need memory whose size is not known in advance to the caller

```
// this is pseudo-C code
char* ReadFile(char* filename) {
  int size = GetFileSize(filename);
  char* buffer = AllocateMem(size);

  ReadFileIntoBuffer(filename, buffer);
  return buffer;
}
```

Dynamic Allocation

- What we want is ***dynamically***-allocated memory
  - Your program explicitly requests a new block of memory
    - The language allocates it at runtime, perhaps with help from OS
  - Dynamically-allocated memory persists until either:
    - Your code explicitly deallocated it (*manual memory management*)
    - A garbage collector collects it (*automatic memory management*)
- C requires you to manually manage memory
  - Gives you more control, but causes headaches

Aside: NULL

- NULL is a memory location that is guaranteed to be invalid
  - In C on Linux, NULL is 0x0 and an attempt to dereference NULL *causes a segmentation fault*
- Useful as an indicator of an uninitialized (or currently unused) pointer or allocation error
  - It's better to cause a segfault than to allow the corruption of memory!

segfault.c

```
int main(int argc, char** argv) {
  int* p = NULL;
  *p = 1;   // causes a segmentation fault
  return 0;
}
```

malloc()

- General usage:

```
var = (type*) malloc(size in bytes)
```

- malloc allocates a block of memory of the requested size
  - Returns a pointer to the first byte of that memory
    - And returns NULL if the memory allocation failed!
  - You should assume that the memory initially contains garbage
  - You'll typically use sizeof to calculate the size you need

```
// allocate a 10-float array
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL) {
  return errcode;
}
...    // do stuff with arr
```

calloc()

- General usage:

```
var = (type*) calloc(num, bytes per element)
```

- Like **malloc**, but also zeros out the block of memory
    - Helpful for shaking out bugs
    - Slightly slower; preferred for non-performance-critical code
    - **malloc** and **calloc** are found in `stdlib.h`

```
// allocate a 10-double array
double* arr = (double*) calloc(10, sizeof(double));
if (arr == NULL) {
  return errcode;
}
...   // do stuff with arr
```

free()

- Usage:

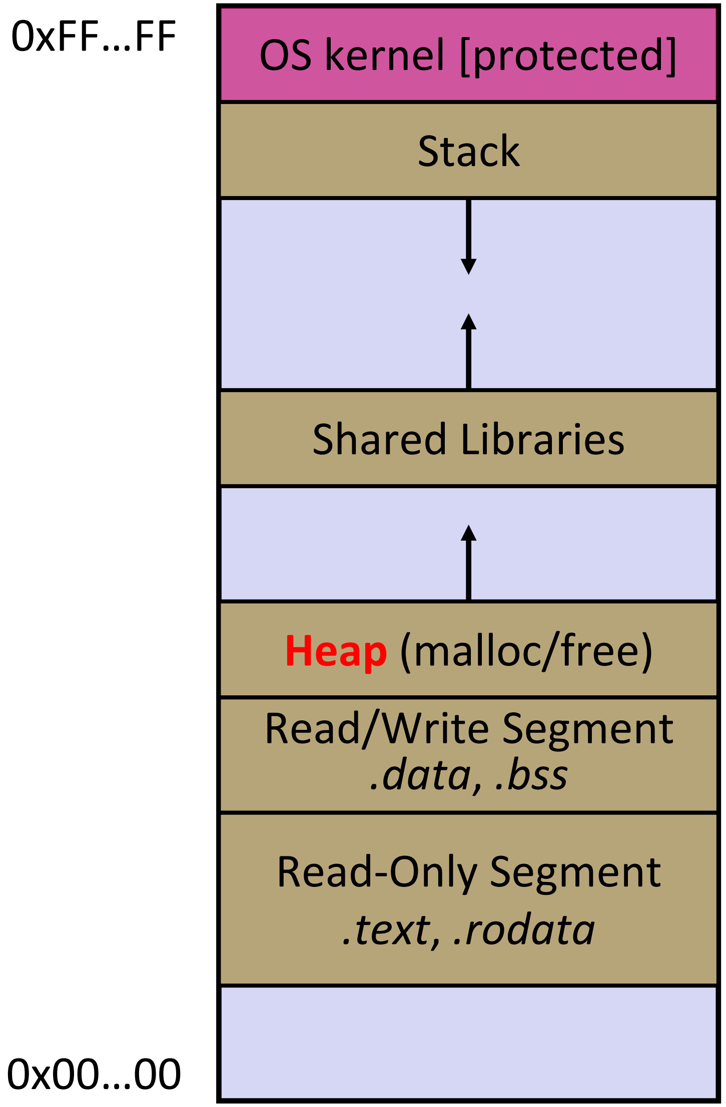
```
free(pointer);
```

- Deallocates the memory pointed-to by the pointer
  - Pointer *must* point to the first byte of heap-allocated memory (*i.e.* something previously returned by **`malloc`** or **`calloc`**)
  - Freed memory becomes eligible for future allocation
  - Pointer is unaffected by call to free
    - Defensive programming: can set pointer to `NULL` after freeing it

```
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL)
  return errcode;
...              // do stuff with arr
free(arr);
arr = NULL;     // OPTIONAL
```

The Heap

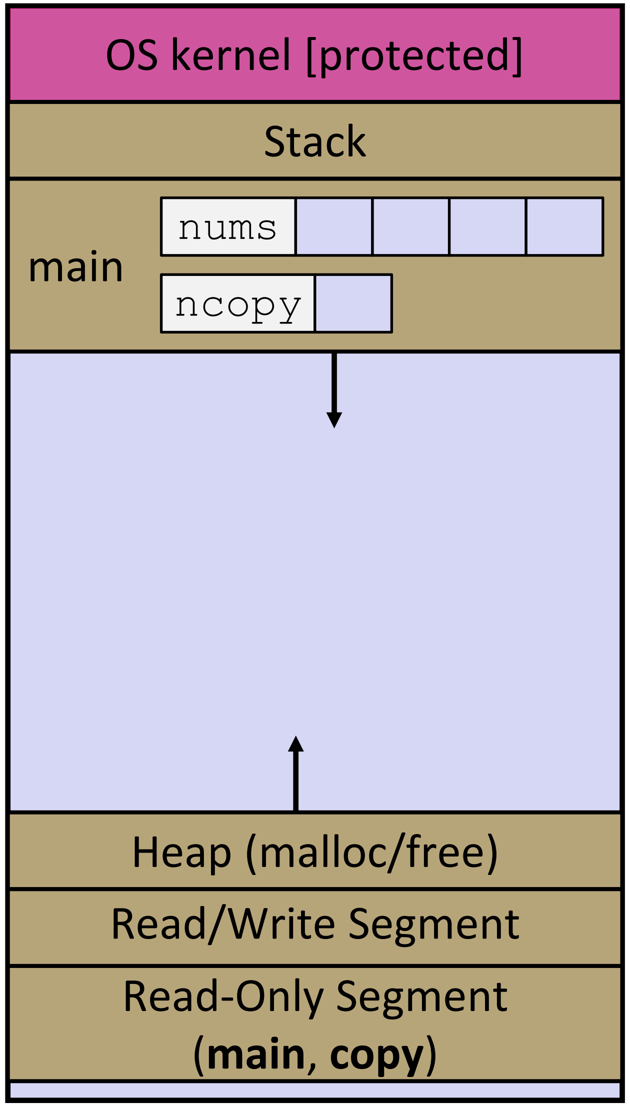
- The Heap is a large pool of unused memory that is used for dynamically-allocated data
  - **malloc** allocates chunks of data in the Heap; **free** deallocates those chunks
  - **malloc** maintains bookkeeping data in the Heap to track allocated blocks



Heap and Stack Example (1/11)

arraycopy.c

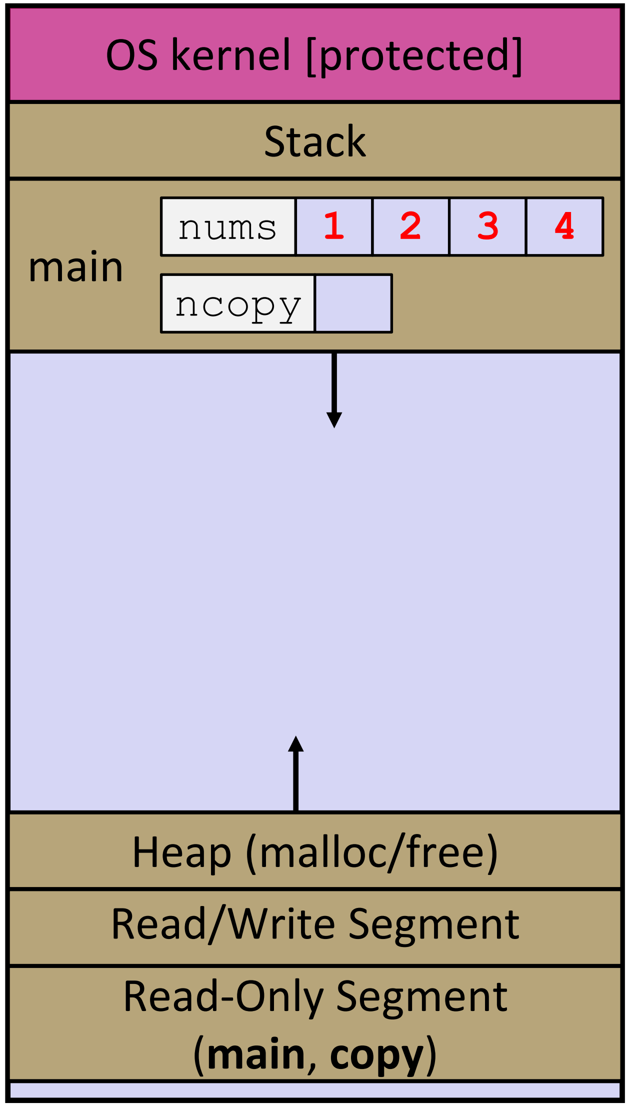
```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example (2/11)

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



OS kernel [protected]

Stack

main

nums 1 2 3 4

ncopy

Heap (malloc/free)

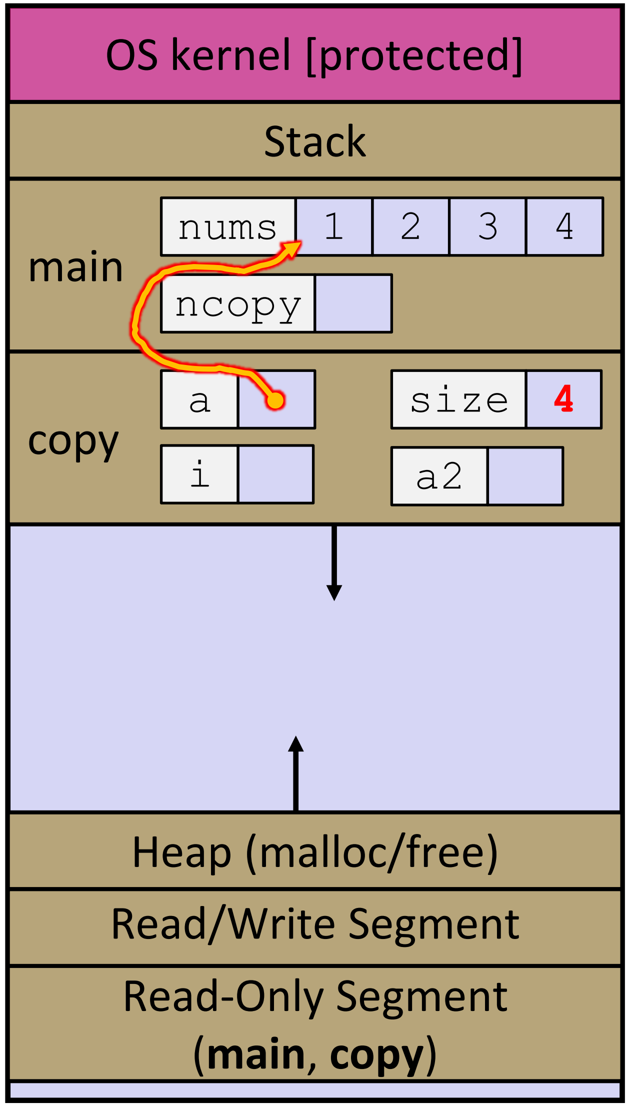
Read/Write Segment

Read-Only Segment
(main, copy)

Heap and Stack Example (3/11)

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



OS kernel [protected]

Stack

main

nums	1	2	3	4

ncopy	

copy

a	

size	4

i	

a2	

Heap (malloc/free)

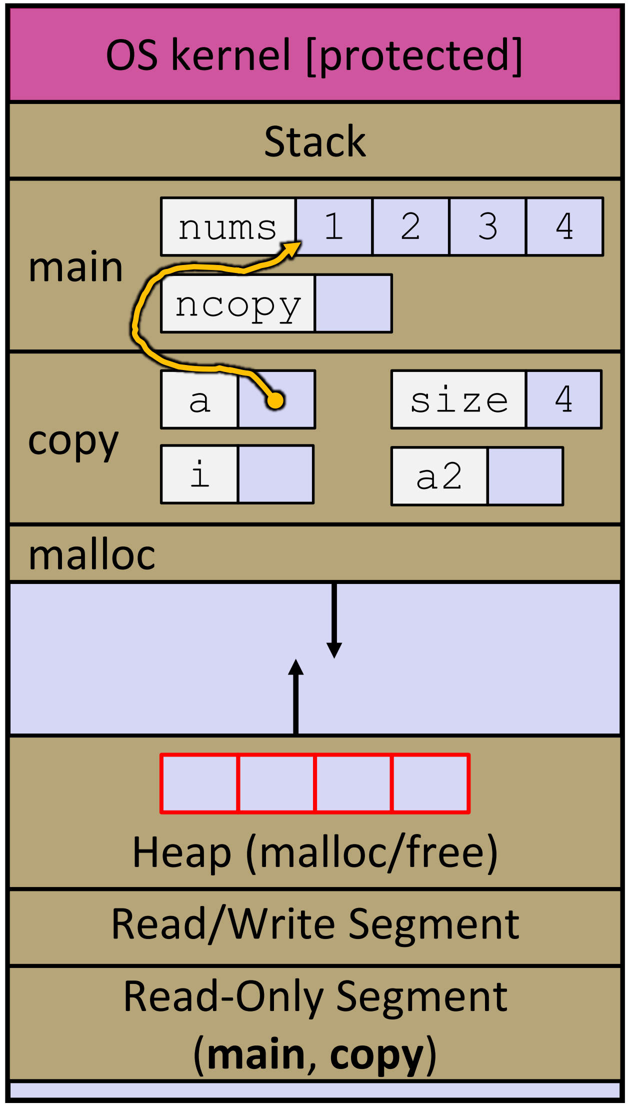
Read/Write Segment

Read-Only Segment
(main, copy)

Heap and Stack Example (4/11)

arraycopy.c

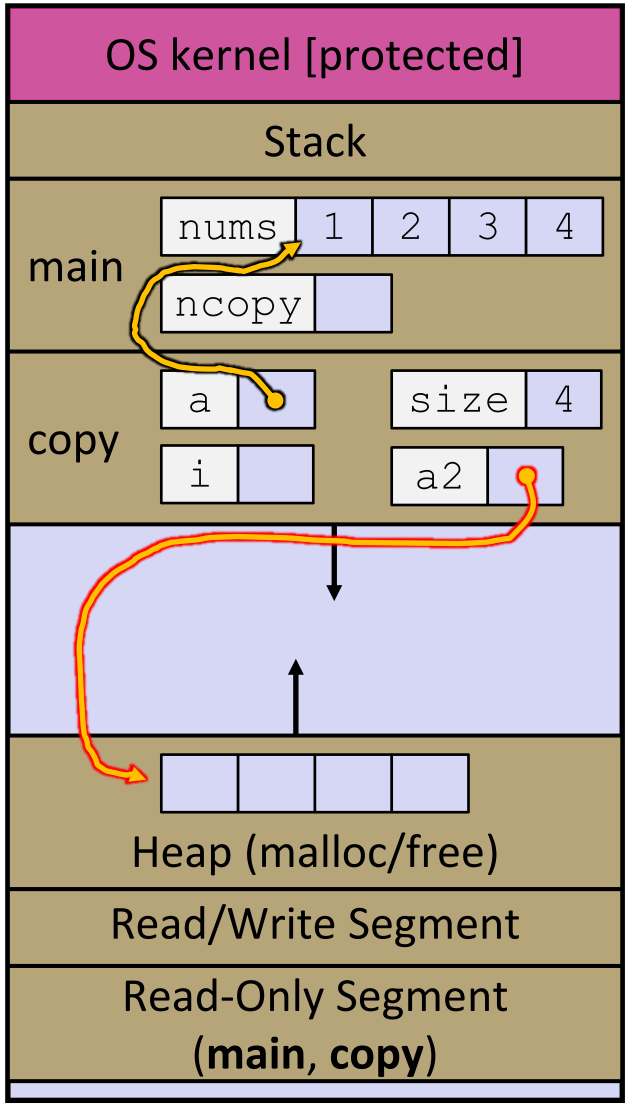
```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example (5/11)

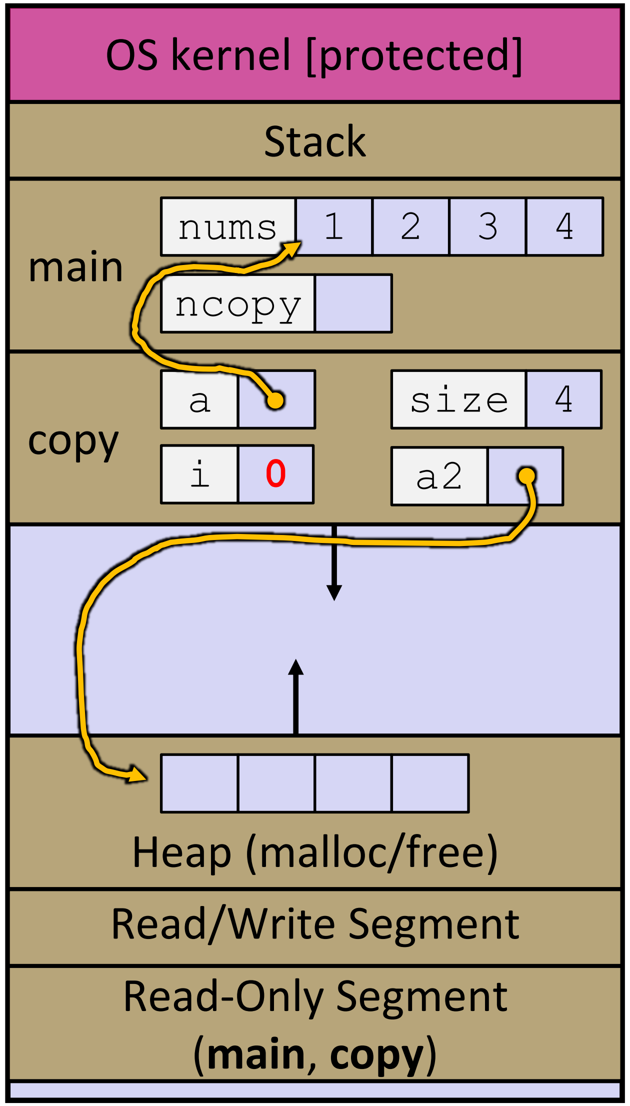
arraycopy.c

```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example (6/11)

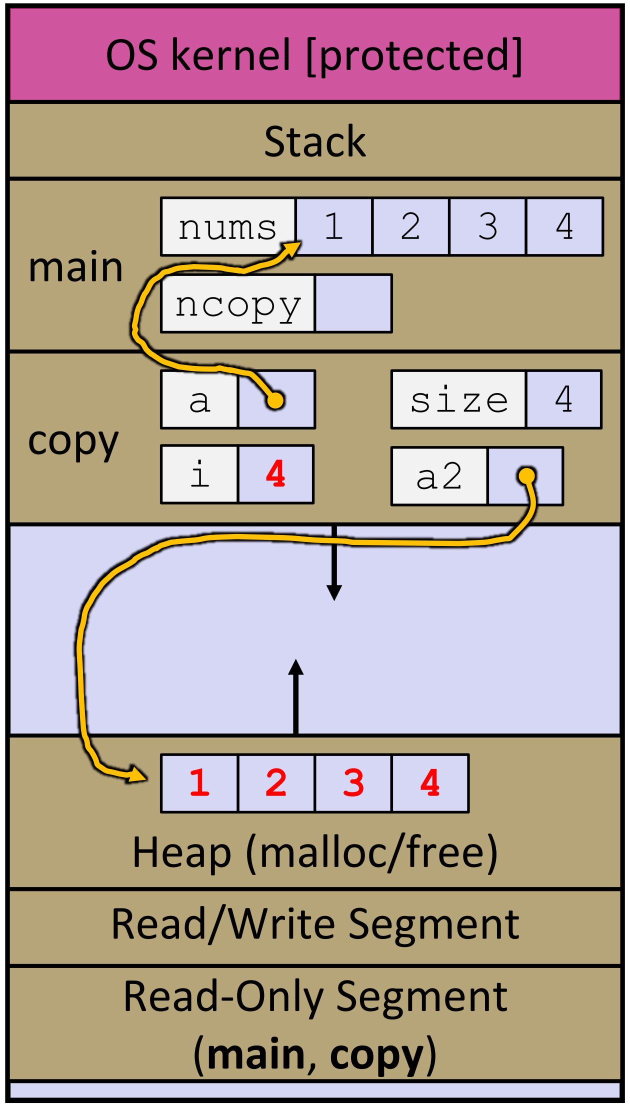
```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example (7/11)

arraycopy.c

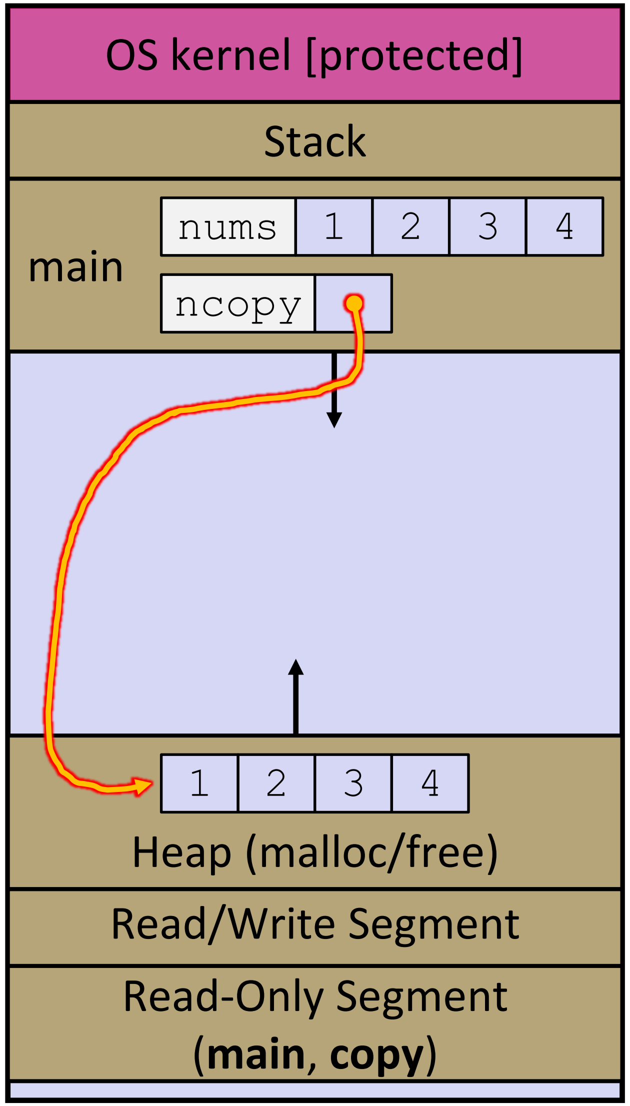
```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example (8/11)

arraycopy.c

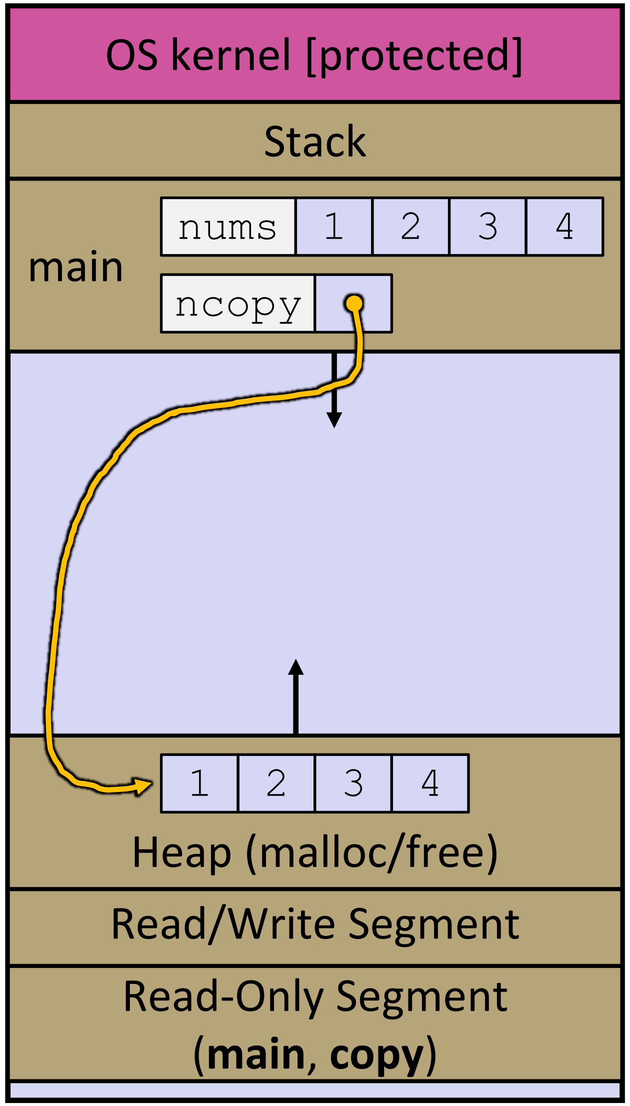
```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example (9/11)

arraycopy.c

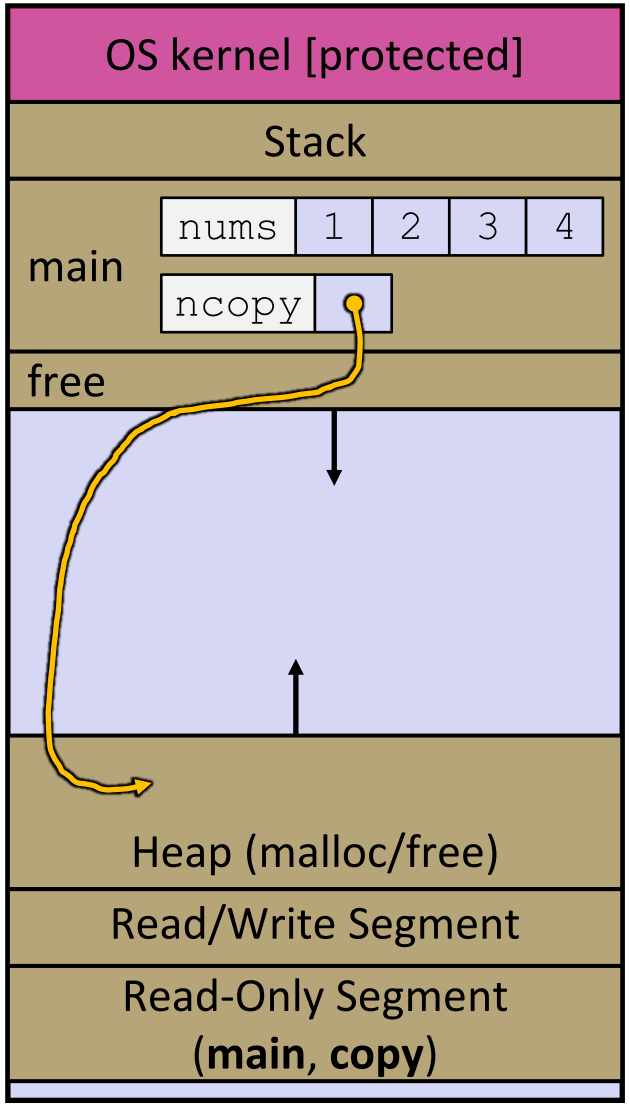
```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Heap and Stack Example (10/11)

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



OS kernel [protected]

Stack

main

nums	1	2	3	4

ncopy

free

Heap (malloc/free)

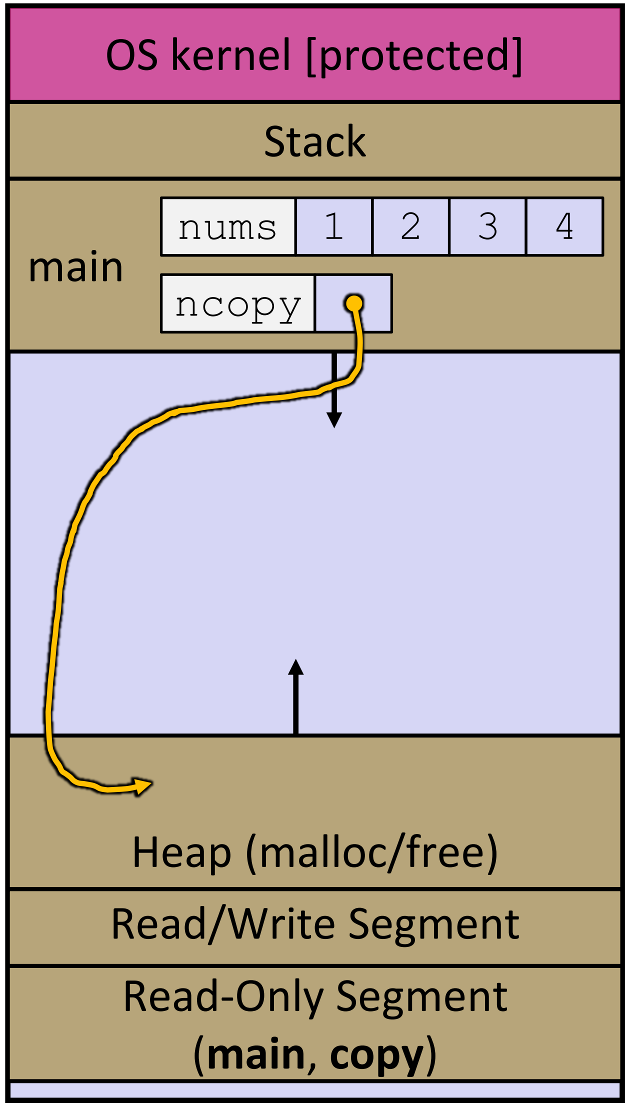
Read/Write Segment

Read-Only Segment
(**main, copy**)

Heap and Stack Example (11/11)

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return 0;
}
```



Memory Corruption

- There are all sorts of ways to corrupt memory in C

memcorrupt.c

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assign past the end of an array
  b[0] += 2;   // assume malloc zeros out memory
  c = b+3;     // mess up your pointer arithmetic
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed pointer

  // any many more!
  return 0;
}
```

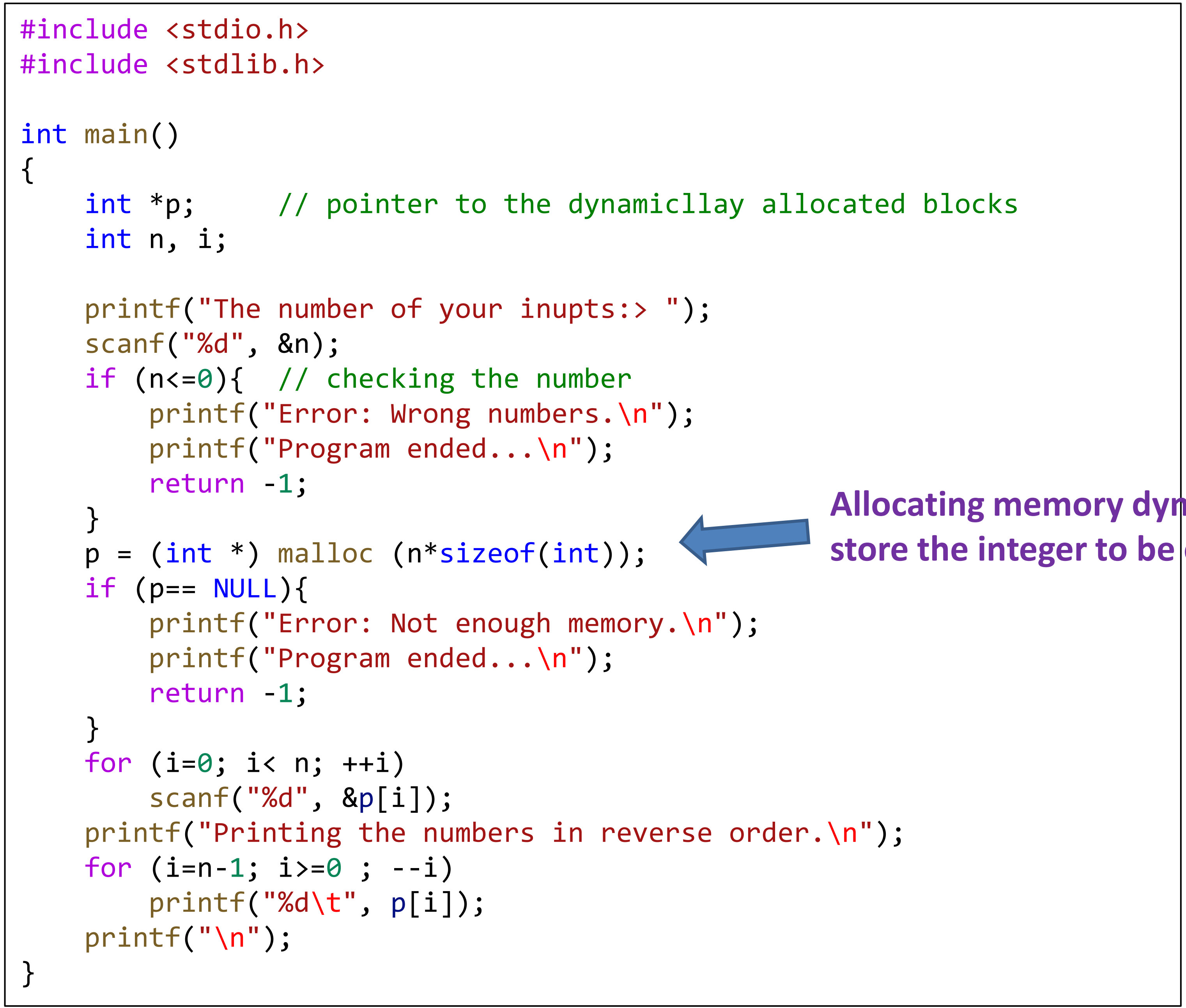
Memory Leak (1/2)

- A memory leak occurs when
  - code fails to deallocate dynamically-allocated memory that is no longer used
  - *e.g.* forget to `free` malloc-ed block, lose/change pointer to malloc-ed block

Memory Leak (2/2)

- Implication: program's VM footprint will keep growing
  - This might be OK for *short-lived* program, since memory deallocated when program ends
  - Usually has bad repercussions for *long-lived* programs
    - Might slow down over time (*e.g.* lead to VM thrashing)
    - Might exhaust all available memory and crash
    - Other programs might get starved of memory

Ex: malloctest.c



```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *p;     // pointer to the dynamicllay allocated blocks
    int n, i;

    printf("The number of your inupts:> ");
    scanf("%d", &n);
    if (n<=0){  // checking the number
        printf("Error: Wrong numbers.\n");
        printf("Program ended...\n");
        return -1;
    }
    p = (int *) malloc (n*sizeof(int));
    if (p== NULL){
        printf("Error: Not enough memory.\n");
        printf("Program ended...\n");
        return -1;
    }
    for (i=0; i< n; ++i)
        scanf("%d", &p[i]);
    printf("Printing the numbers in reverse order.\n");
    for (i=n-1; i>=0 ; --i)
        printf("%d\t", p[i]);
    printf("\n");
}
```

Allocating memory dynamically to store the integer to be entered

C library macro - assert()

- **void assert(int expression)**
  - allows diagnostic information to be written to the standard error file.
  - **expression** – This can be a variable or any C expression
    - evaluates to TRUE, assert() does nothing.
    - evaluates to FALSE, assert() displays an error message on **stderr** (standard error stream to display error messages and diagnostics) and aborts program execution.

Ex: astest.c

```c
#include <stdio.h>
#include <string.h>
#include <assert.h>     // assert() is defined

void copy(char *dest, char *src)
{
    assert(dest != NULL);    // if dest==NULL, then abort
    assert(src != NULL);     // if src==NULL, then abort
    strcpy(dest, src);       // copy string
}

int main()
{
    char s1[100];
    char *s2 = "Hello, world!";

    copy(s1, s2);     // normal execuation

    copy(NULL, s2);   // src is NULL
    // Assertion failed: dest != NULL,
    return 0;
}
```

```
$ gcc –o astest astest.c
$ ./astest
astest: astest.c:7: copy:
        Assertion `dest != NULL' failed.
Aborted (core dumped)
```

Q&A